A Minicourse on Dynamic Multithreaded Algorithms

Charles E. Leiserson*
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, Massachusetts 02139, USA

December 5, 2005

Abstract

This tutorial teaches dynamic multithreaded algorithms using a Cilk-like [11, 8, 10] model. The material was taught in the MIT undergraduate class *6.046 Introduction to Algorithms* as two 80-minute lectures. The style of the lecture notes follows that of the textbook by Cormen, Leiserson, Rivest, and Stein [7], but the pseudocode from that textbook has been "Cilkified" to allow it to describe multithreaded algorithms. The first lecture teaches the basics behind multithreading, including defining the measures of work and critical-path length. It culminates in the greedy scheduling theorem due to Graham and Brent [9, 6]. The second lecture shows how parallel applications, including matrix multiplication and sorting, can be analyzed using divide-and-conquer recurrences.

1 Dynamic multithreaded programming

As multiprocessor systems have become increasingly available, interest has grown in parallel programming. ***Multithreaded programming*** is a programming paradigm in which a single program is broken into multiple ***threads*** of control which interact to solve a single problem. These notes provide an introduction to the analysis of "dynamic" multithreaded algorithms, where threads can be created and destroyed as easily as an ordinary subroutine can be called and return.

1.1 Model

Our model of dynamic multithreaded computation is based on the procedure abstraction found in virtually any programming language. As an example, the procedure FIB gives a multithreaded algorithm for computing the Fibonacci numbers:[1]

*Support was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0270, by the National Science Foundation under Grants EIA-9975036 and ACI-0324974, and by the Singapore-MIT Alliance.

[1]This algorithm is a terrible way to compute Fibonacci numbers, since it runs in exponential time when logarithmic methods are known [7, pp. 902–903], but it serves as a good didactic example.

```
FIB($n$)
1  if $n < 2$
2     then return $n$
3  $x \leftarrow$ spawn FIB($n-1$)
4  $y \leftarrow$ spawn FIB($n-2$)
5  sync
6  return $(x+y)$
```

A spawn is the parallel analog of an ordinary subroutine call. The keyword **spawn** before the subroutine call in line 3 indicates that the subprocedure FIB$(n-1)$ can execute in parallel with the procedure FIB(n) itself. Unlike an ordinary function call, however, where the parent is not resumed until after its child returns, in the case of a spawn, the parent can continue to execute in parallel with the child. In this case, the parent goes on to spawn FIB$(n-2)$. In general, the parent can continue to spawn off children, producing a high degree of parallelism.

A procedure cannot safely use the return values of the children it has spawned until it executes a **sync** statement. If any of its children have not completed when it executes a **sync**, the procedure suspends and does not resume until all of its children have completed. When all of its children return, execution of the procedure resumes at the point immediately following the **sync** statement. In the Fibonacci example, the **sync** statement in line 5 is required before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before each had been computed.

The **spawn** and **sync** keywords specify ***logical*** parallelism, not "actual" parallelism. That is, these keywords indicate which code may possibly execute in parallel, but what actually runs in parallel is determined by a ***scheduler***, which maps the dynamically unfolding computation onto the available processors.

We can view a multithreaded computation in graph-theoretic terms as a dynamically unfolding dag $G = (V, E)$, as is shown in Figure 1 for FIB. We define a ***thread*** to be a maximal sequence of instructions not containing the parallel control statements **spawn** , **sync**, and **return** . Threads make up the set V of vertices of the multithreaded computation dag G. Each procedure execution is a linear chain of threads, each of which is connected to its successor in the chain by a ***continuation*** edge. When a thread u spawns a thread v, the dag contains a ***spawn*** edge $(u, v) \in E$, as well as a continuation edge from u to u's successor in the procedure. When a thread u returns, the dag contains an edge (u, v), where v is the thread that immediately follows the next **sync** in the parent procedure. Every computation starts with a single ***initial thread*** and (assuming that the computation terminates), ends with a single ***final thread***. Since the procedures are organized in a tree hierarchy, we can view the computation as a dag of threads embedded in the tree of procedures.

1.2 Performance Measures

Two performance measures suffice to gauge the theoretical efficiency of multithreaded algorithms. We define the ***work*** of a multithreaded computation to be the total time to execute all the operations in the computation on one processor. We define the ***critical-path length*** of a computation to be the longest time to execute the threads along any path of dependencies in the dag. Consider, for

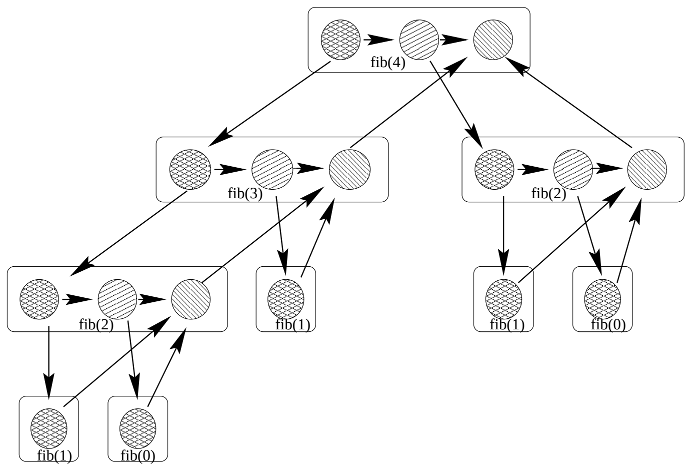

Figure 1: A dag representing the multithreaded computation of $\text{FIB}(4)$. Threads are shown as circles, and each group of threads belonging to the same procedure are surrounded by a rounded rectangle. Downward edges are spawns dependencies, horizontal edges represent continuation dependencies within a procedure, and upward edges are return dependencies.

example, the computation in Figure 1. Suppose that every thread can be executed in unit time. Then, the work of the computation is 17, and the critical-path length is 8.

When a multithreaded computation is executed on a given number P of processors, its running time depends on how efficiently the underlying scheduler can execute it. Denote by T_P the running time of a given computation on P processors. Then, the work of the computation can be viewed as T_1, and the critical-path length can be viewed as T_∞.

The work and critical-path length can be used to provide lower bounds on the running time on P processors. We have

$$T_P \geq T_1/P \,, \tag{1}$$

since in one step, a P-processor computer can do at most P work. We also have

$$T_P \geq T_\infty \,, \tag{2}$$

since a P-processor computer can do no more work in one step than an infinite-processor computer.

The ***speedup*** of a computation on P processors is the ratio T_1/T_P, which indicates how many times faster the P-processor execution is than a one-processor execution. If $T_1/T_P = \Theta(P)$, then we say that the P-processor execution exhibits ***linear speedup***. The maximum possible speedup is T_1/T_∞, which is also called the ***parallelism*** of the computation, because it represents the average amount of work that can be done in parallel for each step along the critical path. We denote the parallelism of a computation by $\overline{P}$.

1.3 Greedy Scheduling

The programmer of a multithreaded application has the ability to control the work and critical-path length of his application, but he has no direct control over the scheduling of his application on a given number of processors. It is up to the runtime scheduler to map the dynamically unfolding computation onto the available processors so that the computation executes efficiently. Good online schedulers are known [3, 4, 5] but their analysis is complicated. For simplicity, we'll illustrate the principles behind these schedulers using an off-line "greedy" scheduler.

A ***greedy scheduler*** schedules as much as it can at every time step. On a P-processor computer, time steps can be classified into two types. If there are P or more threads ready to execute, the step is a ***complete*** step, and the scheduler executes any P threads of those ready to execute. If there are fewer than P threads ready to execute, the step is an ***incomplete*** step, and the scheduler executes all of them. This greedy strategy is provably good.

Theorem 1 (Graham [9], Brent [6]) *A greedy scheduler executes any multithreaded computation G with work T_1 and critical-path length T_∞ in time*

$$T_P \le T_1/P + T_\infty \tag{3}$$

on a computer with P processors.

Proof. For each complete step, P work is done by the P processors. Thus, the number of complete steps is at most T_1/P, because after T_1/P such steps, all the work in the computation has been performed. Now, consider an incomplete step, and consider the subdag G' of G that remains to be executed. Without loss of generality, we can view each of the threads executing in unit time, since we can replace a longer thread with a chain of unit-time threads. Every thread with in-degree 0 is ready to be executed, since all of its predecessors have already executed. By the greedy scheduling policy, all such threads are executed, since there are strictly fewer than P such threads. Thus, the critical-path length of G' is reduced by 1. Since the critical-path length of the subdag remaining to be executed decreases by 1 each for each incomplete step, the number of incomplete steps is at most T_∞. Each step is either complete or incomplete, and hence Inequality (3) follows. $\square$

Corollary 2 *A greedy scheduler achieves linear speedup when $P = O(\overline{P})$.*

Proof. Since $\overline{P} = T_1/T_\infty$, we have $P = O(T_1/T_\infty)$, or equivalently, that $T_\infty = O(T_1/P)$. Thus, we have $T_P \le T_1/P + T_\infty = O(T_1/P)$. $\square$

1.4 Cilk and ⋆Socrates

Cilk [4, 11, 10] is a parallel, multithreaded language based on the serial programming language C. Instrumentation in the Cilk scheduler provides an accurate measure of work and critical path. Cilk's randomized scheduler provably executes a multithreaded computation on a P-processor computer in $T_P = T_1/P + O(T_\infty)$ expected time. Empirically, the scheduler achieves $T_P \approx T_1/P + T_\infty$ time, yielding near-perfect linear speedup if $P \ll \overline{P}$.

Among the applications that have been programmed in Cilk are the ⋆Socrates and Cilkchess chess-playing programs. These programs have won numerous prizes in international competition and are considered to be among the strongest in the world. An interesting anomaly occurred during the development of ⋆Socrates which was resolved by understanding the measures of work and critical-path length.

The ⋆Socrates program was initially developed on a 32-processor computer at MIT, but it was intended to run on a 512-processor computer at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. A clever optimization was proposed which, during testing at MIT, caused the program to run much faster than the original program. Nevertheless, the optimization was abandoned, because an analysis of work and critical-path length indicated that the program would actually be slower on the NCSA machine.

Let us examine this anomaly in more detail. For simplicity, the actual timing numbers have been simplified. The original program ran in $T_{32} = 65$ seconds at MIT on 32 processors. The "optimized" program ran in $T'_{32} = 40$ seconds also on 32 processors. The original program had work $T_1 = 2048$ seconds and critical-path length $T_\infty = 1$ second. Using the formula $T_P = T_1/P + T_\infty$ as a good approximation of runtime, we discover that indeed $T_{32} = 65 = 2048/32 + 1$. The "optimized" program had work $T_1 = 1024$ seconds and critical-path length $T_\infty = 8$ seconds, yielding $T'_{32} = 40 = 1024/32 + 8$. But, now let us determine the runtimes on 512 processors. We have $T_{512} = 2048/512 + 1 = 5$ and $T'_{512} = 1024/512 + 8 = 10$, which is twice as slow! Thus, by using work and critical-path length, we can predict the performance of a multithreaded computation.

Exercise 1-1. Sketch the multithreaded computation that results from executing FIB(5). Assume that all threads in the computation execute in unit time. What is the work of the computation? What is the critical-path length? Show how to schedule the dag on 2 processors in a greedy fashion by labeling each thread with the time step on which it executes.

Exercise 1-2. Consider the following multithreaded procedure SUM for pairwise adding the elements of arrays $A[1 \,..\, n]$ and $B[1 \,..\, n]$ and storing the sums in $C[1 \,..\, n]$:

```
SUM($A, B, C$)
1  for $i \leftarrow 1$ to length[$A$]
2       do $C[i] \leftarrow$ spawn ADD($A[i], B[i]$)
3  sync
```

```
ADD($x, y$)
1  return ($x + y$)
```

Determine an asymptotic bound on the work, the critical-path length, and the parallelism of the computation in terms of n. Give a divide-and-conquer algorithm for the problem that is as parallel as possible. Analyze your algorithm.

Exercise 1-3. Prove that a greedy scheduler achieves the stronger bound

$$T_P \le (T_1 - T_\infty)/P + T_\infty \; . \tag{4}$$

Exercise 1-4. Prove that the time for a greedy scheduler to execute any multithreaded computation is within a factor of 2 of the time required by an optimal scheduler.

Exercise 1-5. For what number P of processors do the two chess programs described in this section run equally fast?

Exercise 1-6. Professor Tweed takes some measurements of his (deterministic) multithreaded program, which is scheduled using a greedy scheduler, and finds that $T_4 = 80$ seconds and $T_{64} = 10$ seconds. What is the fastest that the professor's computation could possibly run on 10 processors? Use Inequality (4) and the two lower bounds from Inequalities (1) and (2) to derive your answer.

2 Analysis of multithreaded algorithms

We now turn to the design and analysis of multithreaded algorithms. Because of the divide-and-conquer nature of the multithreaded model, recurrences are a natural way to express the work and critical-path length of a multithreaded algorithm. We shall investigate algorithms for matrix multiplication and sorting and analyze their performance.

2.1 Parallel Matrix Multiplication

To multiply two $n \times n$ matrices A and B in parallel to produce a matrix C, we can recursively formulate the problem as follows:

$$\begin{aligned}\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} .\end{aligned}$$

Thus, each $n \times n$ matrix multiplication can be expressed as 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices. The multithreaded procedure MULT multiplies two $n \times n$ matrices, where n is a power of 2, using an auxiliary procedure ADD to add $n \times n$ matrices. This algorithm is not in-place.

```
ADD$(C, T, n)$
 1  if $n = 1$
 2     then $C[1,1] \leftarrow C[1,1] + T[1,1]$
 3  return
 4  partition $C$ and $T$ into $(n/2) \times (n/2)$ submatrices
 5  spawn ADD$(C_{11}, T_{11}, n/2)$
 6  spawn ADD$(C_{12}, T_{12}, n/2)$
 7  spawn ADD$(C_{21}, T_{21}, n/2)$
 8  spawn ADD$(C_{22}, T_{22}, n/2)$
 9  sync
10  return
```

```
MULT$(C, A, B, n)$
 1  if $n = 1$
 2     then $C[1,1] \leftarrow A[1,1] \cdot B[1,1]$
 3  return
 4  allocate a temporary matrix $T[1 \,..\, n, 1 \,..\, n]$
 5  partition $A$, $B$, $C$, and $T$ into $(n/2) \times (n/2)$ submatrices
 6  spawn MULT$(C_{11}, A_{11}, B_{11}, n/2)$
 7  spawn MULT$(C_{12}, A_{11}, B_{12}, n/2)$
 8  spawn MULT$(C_{21}, A_{21}, B_{11}, n/2)$
 9  spawn MULT$(C_{22}, A_{21}, B_{12}, n/2)$
10  spawn MULT$(T_{11}, A_{12}, B_{21}, n/2)$
11  spawn MULT$(T_{12}, A_{12}, B_{22}, n/2)$
12  spawn MULT$(T_{21}, A_{22}, B_{21}, n/2)$
13  spawn MULT$(T_{22}, A_{22}, B_{22}, n/2)$
14  sync
15  ADD$(C, T, n)$
```

The matrix partitionings in line 5 of MULT and line 4 of ADD take $O(1)$ time, since only a constant number of indexing operations are required.

To analyze this algorithm, let $A_P(n)$ be the P-processor running time of ADD on $n \times n$ matrices, and let $M_P(n)$ be the P-processor running time of MULT on $n \times n$ matrices. The work (running time on one processor) for ADD can be expressed by the recurrence

$$\begin{aligned} A_1(n) &= 4A_1(n/2) + \Theta(1) \\ &= \Theta(n^2) \;, \end{aligned}$$

which is the same as for the ordinary double-nested-loop serial algorithm. Since the spawned procedures can be executed in parallel, the critical-path length for ADD is

$$\begin{aligned} A_\infty(n) &= A_\infty(n/2) + \Theta(1) \\ &= \Theta(\lg n) \;. \end{aligned}$$

The work for MULT can be expressed by the recurrence

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + A_1(n) \\ &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \;, \end{aligned}$$

which is the same as for the ordinary triple-nested-loop serial algorithm. The critical-path length for MULT is

$$\begin{aligned} M_\infty(n) &= M_\infty(n/2) + \Theta(\lg n) \\ &= \Theta(\lg^2 n) \;. \end{aligned}$$

Thus, the parallelism for MULT is $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is quite high. To multiply 1000×1000 matrices, for example, the parallelism is (ignoring constants) about $1000^3/10^2 = 10^7$. Most parallel computers have far fewer processors.

To achieve high performance, it is often advantageous for an algorithm to use less space, because more space usually means more time. For the matrix-multiplication problem, we can eliminate the temporary matrix T in exchange for reducing the parallelism. Our new algorithm MULT-ADD performs $C \leftarrow C + A \cdot B$ using a similar divide-and-conquer strategy to MULT.

```
MULT-ADD$(C, A, B, n)$
 1  if $n = 1$
 2     then $C[1,1] \leftarrow C[1,1] + A[1,1] \cdot B[1,1]$
 3  return
 4  partition $A$, $B$, and $C$ into $(n/2) \times (n/2)$ submatrices
 5  spawn MULT-ADD$(C_{11}, A_{11}, B_{11}, n/2)$
 6  spawn MULT-ADD$(C_{12}, A_{11}, B_{12}, n/2)$
 7  spawn MULT-ADD$(C_{21}, A_{21}, B_{11}, n/2)$
 8  spawn MULT-ADD$(C_{22}, A_{21}, B_{12}, n/2)$
 9  sync
10  spawn MULT-ADD$(C_{11}, A_{12}, B_{21}, n/2)$
11  spawn MULT-ADD$(C_{12}, A_{12}, B_{22}, n/2)$
12  spawn MULT-ADD$(C_{21}, A_{22}, B_{21}, n/2)$
13  spawn MULT-ADD$(C_{22}, A_{22}, B_{22}, n/2)$
14  sync
15  return
```

Let $MA_P(n)$ be the P-processor running time of MULT-ADD on $n \times n$ matrices. The work for MULT-ADD is $MA_1(n) = \Theta(n^3)$, following the same analysis as for MULT, but the critical-path length is now

$$\begin{aligned} MA_\infty(n) &= 2MA_\infty(n/2) + \Theta(1) \\ &= \Theta(n) , \end{aligned}$$

since only 4 recursive calls can be executed in parallel.

Thus, the parallelism is $MA_1(n)/MA_\infty(n) = \Theta(n^2)$. On 1000×1000 matrices, for example, the parallelism is (ignoring constants) still quite high: about $1000^2 = 10^6$. In practice, this algorithm often runs somewhat faster than the first, since saving space often saves time due to hierarchical memory.

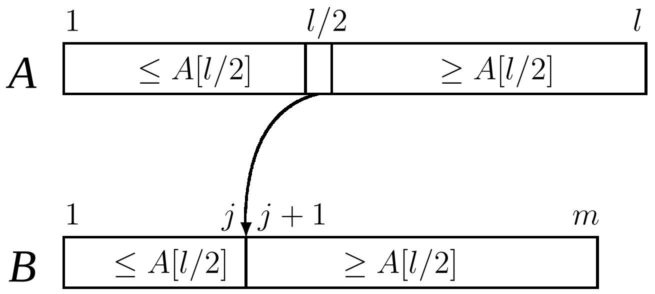

Figure 2: Illustration of P-MERGE. The median of array A is used to partition array B, and then the lower portions of the two arrays are recursively merged, as, in parallel, are the upper portions.

2.2 Parallel Merge Sort

This section shows how to parallelize merge sort. We shall see the parallelism of the algorithm depends on how well the merge subroutine can be parallelized.

The most straightforward way to parallelize merge sort is to run the recursion in parallel, as is done in the following pseudocode:

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2     then $q \leftarrow \lfloor (p + r)/2 \rfloor$
3          spawn MERGE-SORT($A, p, q$)
4          spawn MERGE-SORT($A, q + 1, r$)
5          sync
6          MERGE($A, p, q, r$)
7  return
```

The work of MERGE-SORT on an array of n elements is

$$\begin{aligned} T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \ , \end{aligned}$$

since the running time of MERGE is $\Theta(n)$. Since the two recursive spawns operate in parallel, the critical-path length of MERGE-SORT is

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \Theta(n) \\ &= \Theta(n) \ . \end{aligned}$$

Consequently, the parallelism of the algorithm is $T_1(n)/T_\infty(n) = \Theta(\lg n)$, which is puny. The obvious bottleneck is MERGE.

The following pseudocode, which is illustrated in Figure 2, performs the merge in parallel.

```
P-MERGE$(A[1\,..\,l], B[1\,..\,m], C[1\,..\,n])$
 1  if $m > l$    $\triangleright$ without loss of generality, larger array should be first
 2     then P-MERGE$(B[1\,..\,m], A[1\,..\,l], C[1\,..\,n])$
 3          return
 4  if $n = 1$
 5     then $C[1] \leftarrow A[1]$
 6          return
 7  if $l = 1$    $\triangleright$ and $m = 1$
 8     then if $A[1] \le B[1]$
 9             then $C[1] \leftarrow A[1]$; $C[2] \leftarrow B[1]$
10             else $C[1] \leftarrow B[1]$; $C[2] \leftarrow A[1]$
11          return
12  find $j$ such that $B[j] \le A[l/2] \le B[j+1]$ using binary search
13  spawn P-MERGE$(A[1\,..\,(l/2)], B[1\,..\,j], C[1\,..\,(l/2+j)])$
14  spawn P-MERGE$(A[(l/2+1)\,..\,l], B[(j+1)\,..\,m], C[(l/2+j+1)\,..\,n])$
15  sync
16  return
```

This merging algorithm finds the median of the larger array and uses it to partition the smaller array. Then, the lower portions of the two arrays are recursively merged, and in parallel, so are the upper portions.

To analyze P-MERGE, let $\mathit{PM}_P(n)$ be the P-processor time to merge two arrays A and B having $n = m + l$ elements in total. Without loss of generality, let A be the larger of the two arrays, that is, assume $l \ge m$.

We'll analyze the critical-path length first. The binary search of B takes $\Theta(\lg m)$ time, which in the worst case is $\Theta(\lg n)$. Since the two recursive spawns in lines 13 and 14 operate in parallel, the worst-case critical-path length is $\Theta(\lg n)$ plus the worst-case critical path-length of the spawn operating on the larger subarrays. In the worst case, we must merge half of A with all of B, in which case the recursive spawn operates on at most $3n/4$ elements. Thus, we have

$$\begin{aligned} \mathit{PM}_\infty(n) &\le \mathit{PM}_\infty(3n/4) + \Theta(\lg n) \\ &= \Theta(\lg^2 n)\ . \end{aligned}$$

To analyze the work of MERGE, observe that although the two recursive spawns may operate on different numbers of elements, they always operate on n elements between them. Let αn be the number of elements operated on by the first spawn, where α is a constant in the range $1/4 \le \alpha \le 3/4$. Thus, the second spawn operates on $(1-\alpha)n$ elements, and the worst-case work satisfies the recurrence

$$\mathit{PM}_1(n) = \mathit{PM}_1(\alpha n) + \mathit{PM}_1((1-\alpha)n) + \Theta(\lg n)\ . \tag{5}$$

We shall show that $\mathit{PM}_1(n) = \Theta(n)$ using the substitution method. (Actually, the Akra-Bazzi method [2], if you know it, is simpler.) We assume inductively that $\mathit{PM}_1(n) \le an - b\lg n$ for some constants $a, b > 0$. We have

$$\begin{aligned}
PM_1(n) &\leq a\alpha n - b\lg(\alpha n) + a(1-\alpha)n - b\lg((1-\alpha)n) + \Theta(\lg n) \\
&= an - b(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= an - b(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= an - b\lg n - (b(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\leq an - b\lg n \,,
\end{aligned}$$

since we can choose b large enough so that $b(\lg n + \lg(\alpha(1-\alpha)))$ dominates $\Theta(\lg n)$. Moreover, we can pick a large enough to satisfy the base conditions. Thus, $PM_1(n) = \Theta(n)$, which is the same work asymptotically as the ordinary, serial merging algorithm.

We can now reanalyze the MERGE-SORT using the P-MERGE subroutine. The work $T_1(n)$ remains the same, but the worst-case critical-path length now satisfies

$$\begin{aligned}
T_\infty(n) &= T_\infty(n/2) + \Theta(\lg^2 n) \\
&= \Theta(\lg^3 n) \,.
\end{aligned}$$

The parallelism is now $\Theta(n\lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$.

Exercise 2-1. Give an efficient and highly parallel multithreaded algorithm for multiplying an $n \times n$ matrix A by a length-n vector x that achieves work $\Theta(n^2)$ and critical path $\Theta(\lg n)$. Analyze the work and critical-path length of your implementation, and give the parallelism.

Exercise 2-2. Describe a multithreaded algorithm for matrix multiplication that achieves work $\Theta(n^3)$ and critical path $\Theta(\lg n)$. Comment informally on the locality displayed by your algorithm in the ideal cache model as compared with the two algorithms from this section.

Exercise 2-3. Write a Cilk program to multiply an $n_1 \times n_2$ matrix by an $n_2 \times n_3$ matrix in parallel. Analyze the work, critical-path length, and parallelism of your implementation. Your algorithm should be efficient even if any of n_1, n_2, and n_3 are 1.

Exercise 2-4. Write a Cilk program to implement Strassen's matrix multiplication algorithm in parallel as efficiently as you can. Analyze the work, critical-path length, and parallelism of your implementation.

Exercise 2-5. Write a Cilk program to invert a symmetric and positive-definite matrix in parallel. (*Hint:* Use a divide-and-conquer approach based on the ideas of Theorem 31.12 from [7].)

Exercise 2-6. Akl and Santoro [1] have proposed a merging algorithm in which the first step is to find the median of all the elements in the two sorted input arrays (as opposed to the median of the elements in the larger subarray, as is done in P-MERGE). Show that if the total number of elements in the two arrays is n, this median can be found using $\Theta(\lg n)$ time on one processor in the worst case. Describe a linear-work multithreaded merging algorithm based on this subroutine that has a parallelism of $\Theta(n/\lg^2 n)$. Give and solve the recurrences for work and critical-path length, and determine the parallelism. Implement your algorithm as a Cilk program.

Exercise 2-7. Generalize the algorithm from Exercise 2-6 to find arbitrary order statistics. Describe a merge-sorting algorithm with $\Theta(n \lg n)$ work that achieves a parallelism of $\Theta(n/\lg n)$. (*Hint:* Merge many subarrays in parallel.)

Exercise 2-8. The length of a longest-common subsequence of two length-n sequences x and y can be computed in parallel using a divide-and-conquer multithreaded algorithm. Denote by $c[i,j]$ the length of a longest common subsequence of $x[1\,..\,i]$ and $y[1\,..\,j]$. First, the multithreaded algorithm recursively computes $c[i,j]$ for all i in the range $1 \le i \le n/2$ and all j in the range $1 \le j \le n/2$. Then, it recursively computes $c[i,j]$ for $1 \le i \le n/2$ and $n/2 < j \le n$, while in parallel recursively computing $c[i,j]$ for $n/2 < i \le n$ and $1 \le j \le n/2$. Finally, it recursively computes $c[i,j]$ for $n/2 < i \le n$ and $n/2 < j \le n$. For the base case, the algorithm computes $c[i,j]$ in terms of $c[i-1,j-1]$, $c[i-1,j]$, and $c[i,j-1]$ in the ordinary way, since the logic of the algorithm guarantees that these three values have already been computed.

That is, if the dynamic programming tableau is broken into four pieces

$$\begin{pmatrix} I & II \\ III & IV \end{pmatrix} ,$$

then the recursive multithreaded code would look something like this:

```
$I$
**spawn** $II$
**spawn** $III$
**sync**
$IV$
**return**
```

Analyze the work, critical-path length, and parallelism of this algorithm. Describe and analyze an algorithm that is asymptotically as efficient (same work) but more parallel. Make whatever interesting observations you can. Write an efficient Cilk program for the problem.

References

[1] Selim G. Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11), November 1987.

[2] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Application*, 10:195–210, 1998.

[3] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[6] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, June 1998.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[10] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[11] Supercomputing Technologies Group, MIT Computer Science and Artificial Intelligence Laboratory. *Cilk 5.3.2 Reference Manual*, November 2001.